A Singleton in Port

James W. Cooper

Recently, I took a look at how you communicate with the serial port on your computer in Java. Sun has released an extension package called javax.comm that allows you to ask for ownership of a serial or parallel port, open it and transmit messages with it. You can download this package from the Javasoft site (java.sun.com) for free for Windows and Solaris machines. At this writing it doesn't seem to be available for the Macintosh on the Apple site, but keep looking.

The idea of an object, which you can use to request, open and control a serial port, raises some interesting questions. Obviously, you can't have many instances of such an object active at once, but you'd like to be able to get at the port from any part of your program. The solution to this problem is the Singleton design pattern. A Singleton ensures that there is no more than one instance of a class and provides global access to that instance. Let's start by discussing ways you can implement a Singleton and then see how it was done in the javax.comm package.

Creating Singleton Using a Static Method

Let's say we wanted to use a Singleton to make sure that our system has one and only one print spooler. The easiest way to make a spooler class that can have only one instance is to embed a `static` variable inside the class that we set on the first instance and check for each time we enter the constructor. A static variable is one for which there is only one instance, no matter how many instances there are of the class. To prevent instantiating the class more than once, we make the constructor private so an instance can only be created from within the static method of the class.

Then we create a method called Instance that will return an instance of Spooler, or null if the class has already been instantiated.

```
class iSpooler {
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag = false; //true if 1 instance
//----------------------------------------------
   //the constructor is privatized
   //but need not do anything
   private iSpooler()    {  }
//----------------------------------------------
  //static Instance method returns one instance or null
   static public iSpooler Instance()    {
      if (! instance_flag)       {
         instance_flag = true;      //flag to allow only one
         return new iSpooler();     //only callable from within
      }
      else
         return null;      //return no further instances
   }
   //----------------------------------------------
   public void finalize()    {
      instance_flag = false;
   }
}
```

One major advantage to this approach is that you don't have to worry about exception handling if the Singleton already exists-- you simply get a null return from the Instance method:

```
      iSpooler pr1, pr2;

     //open one spooler--this should always work
      System.out.println("Opening one spooler");
      pr1 = iSpooler.Instance();

     if(pr1 != null)
         System.out.println("got 1 spooler");
      //try to open another spooler --should fail
      System.out.println("Opening two spoolers");

      pr2 = iSpooler.Instance();
      if(pr2 == null)
         System.out.println("no instance available");
```

And, should you try to create instances of the iSpooler class directly, this will fail at compile time because the constructor has been declared as private.

```
//fails at compile time because constructor is privatized
 iSpooler pr3 = new iSpooler();

 static boolean instance_flag = false;
```

Finally, should you need to change the program to allow two or three instances, this class is easily modified to allow this.

Using Exceptions for Singletons

Another way for a Singleton to provide or deny access is by throwing an Exception. It is this approach that the javax.comm package takes. The global point of access is provided through the CommPortIdentifier class, which has the following methods among others:

`String getCurrentOwner()`	Returns owner of the port.
`String getName()`	Returns the name of the port.
`static Enumeration getPortIdentifiers()`	Returns an enumeration of CommPortIdentifiers for the system
`int getPortType()`	Whether serial or parallel
`boolean isCurrentlyOwned()`	Returns whether the port is already owned
`CommPort open(String app, int ss)`	Opens the port, trying for ss msec.

You will note a certain schizophrenia about these methods: one of them is static and it returns an enumeration of other CommPortIdentifier classes. This is the single point of access required for a Singleton. You call it from the class, not from the instance as we show in the example below. Each of the CommPortIdentifier instances that are returned contain a reference to one port. So to find out the number of available serial ports on your PC, you use the following code:

```
Enumeration enum = CommPortIdentifier.getCommPortIdentifiers();
while (portEnum.hasMoreElements()) {
    portId = (CommPortIdentifier) portEnum.nextElement();
    if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL)
         System.out.println(portId.getName());
```

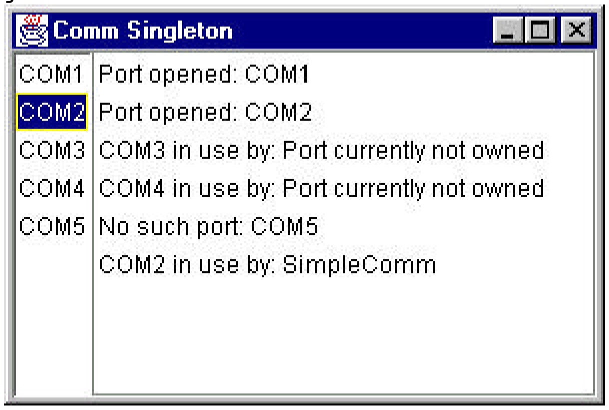
If you do this on your PC, you will find that it always prints out

```
COM1
COM2
COM3
COM4
```

which is usually an overstatement of facts. Apparently the native DLL has engaged in fairly lazy coding and simply asks how many there *could be*, rather than how many there *are*. So the only way to find out if a port actually exists is to try to open it and see if that fails. We can catch the PortInUseException and the NoSuchPortException and print out the appropriate messages. In the example snippet below, you click on a list box of Comm port names and it tries to open whichever one you select and posts a message to the screen:

```
//get port name back from list box
String portName = (String)ports.elementAt (index);
try {
  //try to get port ownership
  portId = CommPortIdentifier.getPortIdentifier(portName);

  //if successful, open the port
  CommPort cp = portId.open("SimpleComm",100);
  //report success
  status.add("Port opened: "+portName);
 }
 catch(NoSuchPortException e){
   status.add("No such port: "+portName);
 }
 catch(PortInUseException e){
   status.add (portName+" in use by: "+portId.getCurrentOwner());
 }
```



```
Port opened: COM1
Port opened: COM2
COM3 in use by: Port currently not owned
COM4 in use by: Port currently not owned
No such port: COM5
COM2 in use by: SimpleComm
```

In the actual program shown in the screen shot, we added a fifth bogus port to see how it would be handled. We clicked on Com1 and Com2 and found that they opened as expected. Clicking on COM3 and COM4 gave back a PortInUseException but no owner rather than a NoSuchPortException. Only COM5 reported the expected no such port exception. However, since a port has to pass both tests to be usable, this is a minor issue. Interestingly, if you get an instance of the COM3 port, the *isCurrentlyOwned* method returns false.

Building a CommPortManager

Let’s consider constructing an enclosing class for these Singleton methods and call it a PortManager. We expect the class to allow us to enumerate the available ports, and open any of them. Some of the methods in such a class might be the ones we show in this parent abstract class below.

```
public abstract class PortManager {
public static Enumeration getAllPorts()
  {return null;}
public static Enumeration getAvailablePorts()
  {return null;}
public static CommPort openPort(String portName)
  {return null;}
public static CommPort openPort()
  {return null;}
}
```

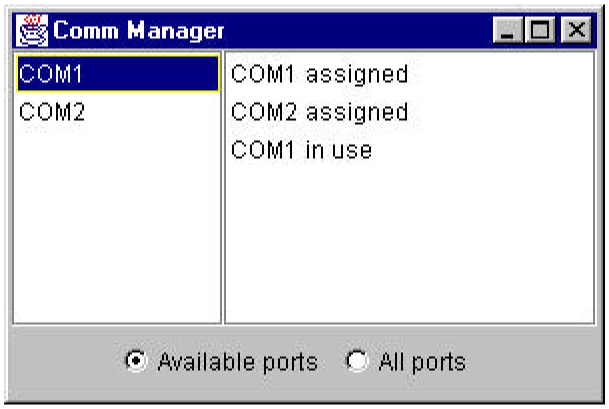
Because of the syntax rules of Java, we can’t have abstract static methods, so we actually declare each of them in the parent abstract class to return *null*.

The *getAllPorts* method is just like the one shown above where we enumerate the ports. However, the *getAvailablePorts* method is interesting because we don’t need to do anything at all when we catch the port exceptions. We get the port and try to open it. If neither operation fails, we add the port name to the list of available ports.

```
public static Enumeration getAvailablePorts() {
   Vector portOpenList = new Vector();
   CommPortIdentifier portId;

   Enumeration enum = getAllPorts();
   while (enum.hasMoreElements ()) {
      String portName = (String)enum.nextElement () ;
      try {
         //try to get port ownership
         portId =
            CommPortIdentifier.getPortIdentifier(portName);
         //if successful, open the port
         CommPort cp = portId.open("SimpleComm",100);
         //report success
         portOpenList.addElement(portName);
         cp.close();
         }
      catch(NoSuchPortException e){} //failure is enough
      catch(PortInUseException e){}
    }
   return portOpenList.elements ();
 }
```

The available ports method is illustrated in the following figure:



```
COM1    COM1 assigned
COM2    COM2 assigned
        COM1 in use
```

[✓ Available ports] All ports

So now at the end, we again might ask, where are the Singletons? We've really just wrapped the CommPortIdentifier methods so we could handle the exceptions internally. There can be any number of instances of our CommPortManager class, but you can only call the static methods from the class rather than from an instance. Thus, they are providing a global point of access to the ports, and there can be only one instance of each open port. The ports themselves remain the Singletons, but the CommPortManager class provides convenient (global) access to these Singletons.

Developing in Java with Visual SlickEdit

Many people who write programs at the edge of new developments in Java find that for various reason, the commercial IDE tools like Symantec Visual Café, Borland JBuilder and IBM's Visual Age for Java are not what they need. Either they can't handle the newest technologies or they impose a complex graphical infrastructure that just isn't needed for code with minimal graphical components, such as Java servers. But these products have a terrific advantage n source code management: they keep all your classes together and assure that all the right modules are compiled as needed. They also have excellent debuggers that help you step through troublesome code. This is hard to give up.

But for simple Java tests of new features and for developing server code, I have been using Visual SlickEdit for some years now. This has to be one of the best programmer's editors out there. It has syntax highlighting and will help you fill in *if* and *for* statements. It does brace completion and will even help you find stray extra or missing braces. You can set it up to compile on one keystroke and can click on error messages to go right to the error spot in the code.

But in Version 4.0 things get even better. VSlick has support for Java projects, so you can keep all your files together and compile only those needed to bring your system up to date. Buts best of all, Vslick will do statement completion. Just type in the name of any class instance followed by a dot, and Vslick will pop up a list of possible methods. And this works for your own classes, too! This is the same seductive feature that makes using Visual Basic such a joy, and I'm making great use of it whenever I introduce new classes into my programming. Thanks to Randy Richardt for showing me these very helpful features.